John von Neumann Institute for Computing

Continuous Runtime Profiling of OpenMP Applications

Karl Fürlinger, Shirley Moore

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 677-684, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Continuous Runtime Profiling of OpenMP Applications

Karl Fürlinger and Shirley Moore

Innovative Computing Laboratory
EECS Department
University of Tennessee
Knoxville, Tennessee, USA
E-mail: {karl, shirley}@eecs.utk.edu

In this paper we investigate the merits of combining tracing and profiling with the goal of limiting data volume and enabling a manual interpretation, while retaining some temporal information about the program execution characteristics. We discuss the general dimensions of performance data and which new kind of performance displays can be derived by adding a temporal dimension to profiling-type data. Among the most useful new displays are *overheads over time* which allows the location of when overheads such as synchronization arise in the target application and *performance counter heatmaps* that show performance counters for each thread over time.

1 Introduction

Profiling and tracing are the two common techniques for performance analysis of parallel applications. Profiling is often preferred over tracing because it generates smaller amounts of data, making a manual interpretation easier. Tracing, on the other hand, allows the full temporal behaviour of the application to be reconstructed at the expense of larger amounts of performance data and an often more intrusive collection process.

In this paper we investigate an approach to combine the advantages of tracing and profiling with the goal of limiting the data volume and enabling manual interpretation, while retaining information about the temporal behaviour of the program. Our starting point is a profiling tool for OpenMP applications called `ompP`[1]. Instead of capturing the profiles only at the end of program execution ("one-shot" profiling), in the new approach profiles are captured at several points of time while the application executes. We call our technique *incremental* or *continuous* profiling and demonstrate its usefulness with a number of examples.

The rest of this paper is organized as follows: Sect. 2 briefly introduces our profiling tool and describes its existing capabilities. Sect. 3 then describes the general dimensions of performance data and the new types of data (often best displayed as graphical views) that become available with continuous profiling. Sect. 4 serves as an evaluation of our idea where we show examples form the SPEC OpenMP benchmark suite[2]. We describe related work in Sect. 5 and conclude and discuss further directions for our work in Sect. 6.

2 Application Profiling with `ompP`

`ompP` is a profiling tool for OpenMP applications designed for Unix-like systems. `ompP` differs from other profiling tools like `gprof` or OProfile[3] in primarily two ways. First, `ompP` is a measurement based profiler and does not use program counter sampling. The

```
R00002 main.c (20-23) (unnamed) CRITICAL
 TID      execT      execC      bodyT     enterT      exitT
   0       1.00          1       1.00       0.00       0.00
   1       3.01          1       1.00       2.00       0.00
   2       2.00          1       1.00       1.00       0.00
   3       4.01          1       1.00       3.01       0.00
 SUM      10.02          4       4.01       6.01       0.00
```

Figure 1. Profiling data delivered by ompP for a critical section for a run with four threads (one line per thread, the last line sums over all threads). All times are durations in seconds.

instrumented application invokes ompP monitoring routines that enable a direct observation of program execution events (like entering or exiting a critical section). An advantage of the direct approach is that its results are not subject to sampling inaccuracy and hence they can also be used for correctness testing in certain contexts.

The second difference is in the way of data collection and representation. While other profilers work on the level of functions, ompP collects and displays performance data in the OpenMP user model of the execution[4]. For example, the data reported for critical section contains not only the execution time but also lists the time to enter and exit the critical construct (enterT and exitT, respectively) as well as the accumulated time each threads spends inside the critical construct (bodyT) and the number of times each thread enters the construct (execC). An example profile of a critical section is given in Fig. 1.

Profiling data in a similar style is delivered for each OpenMP construct, the columns (execution times and counts) depend on the particular construct. Furthermore, ompP supports querying hardware performance counters through PAPI[5] and the measured counter values appear as additional columns in the profiles. In addition to OpenMP constructs that are instrumented automatically using Opari[6], a user can mark arbitrary source code regions such as functions or program phases using a manual instrumentation mechanism.

Profiling data is reported by ompP both as flat profiles as well as callgraph profiles, giving inclusive and exclusive times in the latter case. ompP performs an overhead analysis where four well-defined overhead classes (synchronization, load imbalance, thread management, limited parallelism) are quantitatively evaluated. ompP also tries to detect common inefficiency situations, such as load imbalance in parallel loops, contention for locks and critical sections, etc. The profiling report contains a list of the discovered instances of these – so called – *performance properties*[7] sorted by their severity (negative impact on performance).

3 From Profiling to Continuous Profiling

For both profiling and tracing, the following dimensions of performance data can be distinguished in general:

- Kind of data: describes which type of data is measured or reported to the user. Examples include time stamps or durations, execution counts, performance counter values, and so on.

- Source code location: data can be collected globally (for the entire program) or for specific source code entities such as subroutines, OpenMP constructs, basic blocks, individual statements, etc.

- Thread / process dimension: measured data can either be reported for individual threads or processes or accumulated over groups (by summing or averaging, for example).

- Time dimension: Describes when a particular measurement was made (time-stamp) or for which time duration values have been measured.

A distinguishing and appealing property of profiling data is its low dimensionality, i.e., it can often be comprehended textually (like gprof output) or it can be visualized as 1D or 2D graphs in a straightforward way. Adding a new dimension (time) jeopardizes this advantage and requires more sophisticated performance data management and displays. The following description lists performance data displays from continuous profiles that are based on the (classic) performance data delivered by the `ompP`, extended with a temporal dimension.

- **Performance properties over time:**

  Performance properties[7] are a very compact way to represent performance analysis results and their change over time can thus be visualized easily. There is an extended formalism for specifying properties[7], an informal example for illustration is "Imbalance in parallel region `foo.f (23-42)` with severity of 4.5%". The definition carries all relevant context information with it and the severity value denotes the percentage of total execution time improvement that can be expected if the cause for the inefficiency could be removed. The threads dimension is collapsed in the specification of the property and the source code dimension is encoded as the context of the property (`foo.f (23-42)` in the above example).

  Properties over time data can be visualized as a 1D lineplot, where the x-axis is the time (t) and the y-axis denotes the severity value at time t. That is, the severity value at time t is determined by program behaviour from program start (time 0 until t). Depending on the particular application, valuable information can be deduced from the shapes of the graphs. An example is shown in Fig. 2.

- **Region invocations over time:**

  Depending on the size of the application and the analyst's familiarity with the source code, it can be valuable to know when and how often a particular OpenMP construct, such as a parallel loop, was executed. The region invocation over time displays offers this functionality. This view is most useful when aggregating (e.g., summing) over all threads, the x-axis displays the time and the y-axis counts the region invocations in this case. In certain situations it can also be valuable to see which thread executed a construct at which time. In this case either multiple lineplots (one line per thread) or a surface plot (y-axis representing threads and z-axis counting invocations) can be used for visualization. Another option is colour coding the number of invocations similar to the performance counter heatmap view (discussed below).

- **Region execution time over time:**

  This display is similar to the region invocation over time display but shows the execution time instead of execution count. Again this display allows the developer to see when particular portions of the code actually get executed. In addition, by dividing the execution time by the execution count, a normalized execution time can be determined. This allows a developer to see if the execution time of the region instances changed over time and to derive conclusions from that, e.g., effects like cache pollution can show up in this type of display.

- **Overheads over time:**

  `ompP` evaluates four overhead classes based on the profiling data for individual parallel regions and for the program as a whole. For example, the time required to enter a critical section is attributed as the containing parallel region's synchronization overhead. A detailed discussion and motivation of this classification scheme can be found in[8].

  The overheads over time can be visualized easily as 1D lineplots similar to the properties over time view. The x-axis represents time and the y-axis shows the incurred overhead. It is usually convenient to display the overheads in percentages of execution time lost, i.e., the y-axis ranges from 0 to 100% and for each of the four supported overhead classes (synchronization, imbalance, limited parallelism, thread management), a line indicates the percentage of execution time lost due to that overhead class. Different to the properties over time display, the overheads are plotted as they occur for each time step (Δt) and are not accumulated from program start. An example for this is the graph in Fig. 3.

- **Performance counter heatmaps:**

  The performance counter heatmap display is a tile map where the x-axis corresponds to the time while the y-axis corresponds to the thread ID. The tiles are filled and a colour gradient coding is used to differentiate between higher and lower counter values. A tile is not filled if no data samples are available for that time period. This type of display is supported for both the whole program as well as for individual OpenMP regions.

4 Implementation and Evaluation of Continuous Runtime Profiling

A straightforward way to add a temporal component to profiling-type performance data is to capture profiles at several points during the execution of the target application (and not just at the end) and to analyze how the profiles change between those capture points. Alternatively (and equivalently), the changes between capture points can be recorded incrementally and the overall state at capture time can later be recovered.

Several trigger events for the collection of profiling reports are possible. The trigger can either be based on a fixed-length or adaptive timer, or it can be based on the overflow of a hardware counter. Another possibility is to expose a mechanism to dump profiles to the user. In this paper we investigate the simplest form of incremental profiling: capturing profiles in regular, fixed-length intervals during the entire lifetime of the application. We

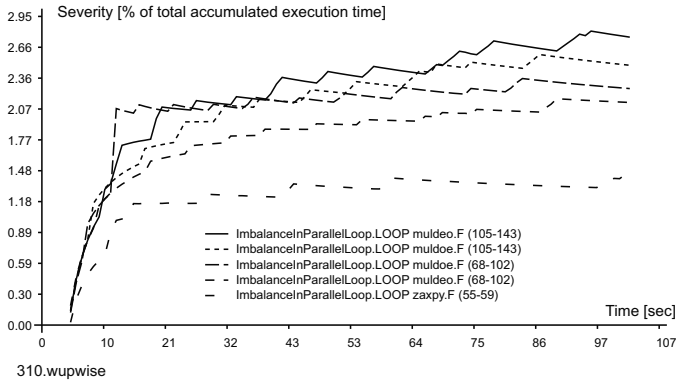


Figure 2. An example for the "performance properties over time" display for the 310.wupwise application. The five most severe performance properties are shown.

have implemented this technique in our profiler `ompP` by registering a timer signal (using `SIGALRM`) that is delivered to the profiler in regular intervals and causes the current state of the profiling data to be stored to a memory buffer. On program termination the buffer is flushed to disk as a set of profiling reports. A collection of Perl scripts that come with `ompP` can then be used to analyze the profiling reports and create the performance displays described below in the form of SVG (scalable vector graphics) and PNG (portable network graphics) images.

We have tested this technique with a dumping interval of 1 second on the applications from the medium size variant of the SPEC OpenMP benchmarks on a 32 CPU SGI Altix machine based on Itanium-2 processors with 1.6 GHz and 6 MB L3 cache used in batch mode. Due to space limitations we can only show results from a very small number of runs.

Fig. 2 shows the properties over time display for the 310.wupwise application. It is evident that the severity of the properties which are all imbalance related appears to be continuously increasing as time proceeds, indicating that the imbalance situations in this code will become increasingly significant with longer runtime (e.g., larger data sets or more iterations). Other applications from the SPEC OpenMP benchmark suite showed other interesting features such as initialization routines that generated high initial overheads which amortized over time (i.e., the severity decreased).

Figure 3 shows the overheads over time display for the 328.fma3d application. The most noticeable overhead is synchronization overhead starting at about 30 seconds of execution and lasting for several seconds. A closer examination of the profiling reports reveals that this overhead is caused by critical section contention. One thread after the other enters the critical section and performs a time-consuming initialization operation. This effectively serializes the execution for more than 10 seconds and shows up as an overhead of $31/32 = 97\%$ in the overheads graph.

The graphs in Fig. 4 show examples of performance counter heatmaps. Depending on the selected hardware counters, this view offers very interesting insight into the behaviour of the applications. Phenomena that we were able to identify with this kind of display

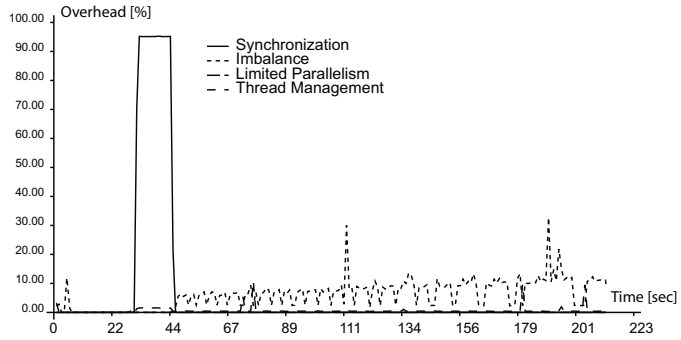


Figure 3. This graph shows overheads over time for the 328.fma3d application.

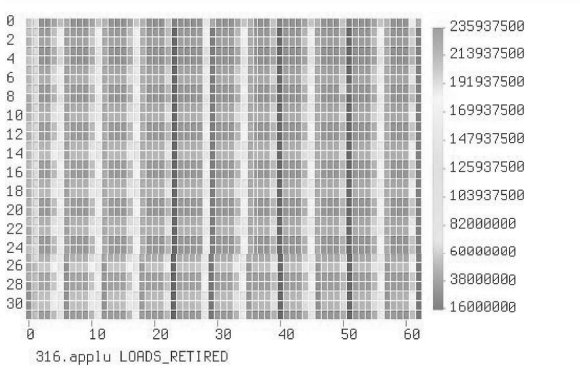
include iterative behaviour (e.g.,Fig. 4(a)), thread grouping and differences in homogeneity or heterogeneity of thread behaviour. E.g., often groups of threads would show markedly different behaviour compared to other threads in a 32 thread run. Possible reasons for this difference in behaviour might be in the application itself (related to the algorithm) but they could also come from the machine organization or system software layer (mapping of threads to processors and their arrangement in the machine and its interconnect). As another example, Fig. 4(b) gives the number of retired floating point operations for the 324.apsi application and this graph shows a marked difference for threads 0 to 14 vs. 15 to 31. We were not able to identify the exact cause for this behaviour yet.

5 Related Work

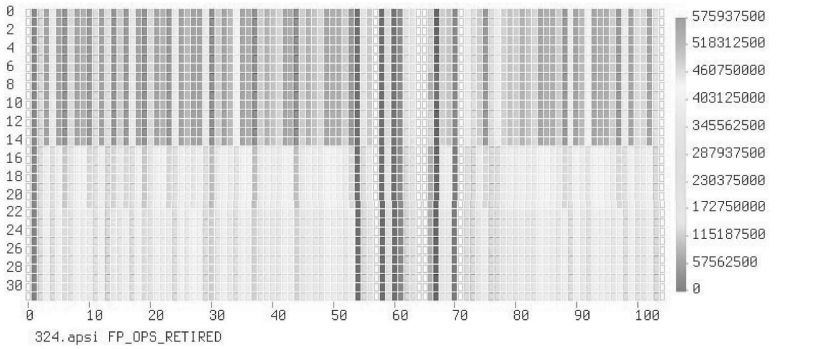
There are several performance analysis tools for OpenMP. Vendor specific tools such as the Intel Thread Profiler and Sun Studio are limited to a single platform but can take greater advantage of internal details of the compiler's OpenMP implementation and the runtime system than more general tools. Both the Intel and the Sun tool are based on sampling and can provide the user with some timeline profile displays. Neither of those tools however has a concept similar to `ompP`'s high-level abstraction of performance properties or the properties over time display.

TAU[9, 10] is also able to profile and trace OpenMP applications by utilizing the Opari instrumenter. Its performance data visualizer Paraprof supports a number of different profile displays and also supports interactive 3D exploration of performance data, but does not currently have a view similar to the performance counter heatmaps. The TAU toolset also contains a utility to convert TAU trace files to profiles which can generate profile series and interval profiles.

OProfile and its predecessor, the Digital Continuous Profiling Infrastructure (DCPI), are system-wide statistical profilers based on hardware counter overflows. Both approaches rely on a profiling daemon running in the background and both support the dumping of profiling reports at any time. Data acquisition in a style similar to our incremental profiling approach would thus be easy to implement. We are, however, not aware of any study



(a) Retired load instructions for the 316.applu application.



(b) Retired floating point operations for the 324.apsi application.

Figure 4. Example performance counter heatmaps. Time is displayed on the horizontal axis (in seconds), the vertical axis lists the threads (32 in this case).

using OProfile or DPCI that investigated continuous profiling for parallel applications. In practice, the necessity of root privileges and the difficulty of relating profiling data back to the user's OpenMP execution model can be a major problem employing these approaches, both are no issues with `ompP` since it is based on source code instrumentation.

6 Outlook and Future Work

We have investigated continuous profiling of parallel applications in the context of an existing profiling tool for OpenMP applications. We have discussed several general approaches to add temporal dimension to performance data and have tested our ideas on applications from the SPEC OpenMP benchmarks suite.

Our results indicate that valuable information about the temporal behaviour of applications can be discovered by incremental profiling and that this technique strikes a good balance between the level of detail offered by tracing and the simplicity and efficiency of profiling. Using continuous profiling we were able to get new insights into the behaviour

of applications which can, due to the lack of temporal data, not be gathered from traditional "one-shot" profiling. The most interesting features are the detection of iterative behaviour, the identification of short-term contention for resources, and the temporal localization of overheads and execution patterns.

We plan continued work in several areas. In a future release of `ompP` we plan to support other triggers for capturing profiles, most importantly user-added and overflow based. Furthermore we intend to test our approach in the context of MPI as well, a planned integrated MPI/OpenMP profiling tool based on `mpiP`[11] and `ompP` is the first step in this direction.

References

1. K. Fuerlinger and M. Gerndt, *ompP: a profiling tool for OpenMP*, in: Proc. 1st International Workshop on OpenMP (IWOMP 2005), Eugene, Oregon, USA, (2005).
2. V. Aslot and R. Eigenmann, *Performance characteristics of the SPEC OMP2001 benchmarks*, SIGARCH Comput. Archit. News, **29**, 31–40, (2001).
3. J. Levon, *OProfile, A system-wide profiler for Linux systems*, Homepage: `http://oprofile.sourceforge.net`.
4. M. Itzkowitz, O. Mazurov, N. Copty and Y. Lin, *An OpenMP runtime API for profiling*, Accepted by the OpenMP ARB as an official ARB White Paper; available online at `http://www.compunity.org/futures/omp-api.html`.
5. S. Browne, J. Dongarra, N. Garner, G. Ho and P. J. Mucci, *A portable programming interface for performance evaluation on modern processors*, Int. J. High Perform. Comput. Appl., **14**, 189–204, (2000).
6. B. Mohr, A. D. Malony, S. S. Shende and F. Wolf, *Towards a performance tool interface for OpenMP: an approach based on directive rewriting*, in: Proc. Third Workshop on OpenMP (EWOMP'01), (2001).
7. M. Gerndt and K. Fürlinger, *Specification and detection of performance problems with ASL*, Concurrency and Computation: Practice and Experience, **19**, 1451–1464, (2007).
8. K. Fuerlinger and M. Gerndt, *Analyzing overheads and scalability characteristics of OpenMP applications*, in: Proc. 7th International Meeting on High Performance Computing for Computational Science (VECPAR'06), Rio de Janeiro, Brasil, LNCS vol. 4395, pp. 39–51, (2006).
9. S. S. Shende and A. D. Malony, *The TAU parallel performance system*, International Journal of High Performance Computing Applications, ACTS Collection Special Issue, (2005).
10. A. D. Malony, S. S. Shende, R. Bell, K. Li, L. Li and N. Trebon, *Advances in the TAU performance analysis system*, in: Performance Analysis and Grid Computing, V. Getov, M. Gerndt, A. Hoisie, A. Malony, and B. Miller, (Eds.), pp. 129–144, (Kluwer 2003).
11. J. S. Vetter and F. Mueller, *Communication characteristics of large-scale scientific applications for contemporary cluster architectures*, J. Parallel Distrib. Comput., **63**, 853–865, (2003).